PLUG-AND-PLAY FOSS ML ACCELERATOR : FROM CONCEPT TO CONCEPTION

A Dissertation
Presented to
The Academic Faculty

By

Yehowshua Immanuel

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering
College of Engineering

Georgia Institute of Technology

Dec 2020

PLUG-AND-PLAY FOSS ML ACCELERATOR : FROM CONCEPT TO CONCEPTION

Thesis committee:

Dr. Tushar Krishna, Advisor
School of Electrical and Compute Engineering
Georgia Institute of Technology

Dr. Saibal Mukhopadhyay
School of Electrical and Compute Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Date approved: December 3, 2020

Do you see a man who excels in his work? He will stand before kings; He will not stand before unknown men.

Proverbs 4:22

For my sister Tehilla

ACKNOWLEDGMENTS

I would like thank my advisor Dr. Krishna for inviting to work on the end-to-end accelerator idea. I would like to thank my Dad and Mom for pushing me to finish my masters. I would like to thank my fellow lab-mates Anand, Eric, and Matthew for providing key insight during the development of different aspects of the MAERI[1] accelerator implementation. I would also like to thank David Yue who helped me on test out certain portions of the MAERI compiler. I'd like to thank Hyoukjun who helped me build early revisions of the hardware. I'd like to thank Geonhwa who help me build early revisions of the compiler. Lastly, I'd like to thank Christ for the strength to press on.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

ML accelerators are a fairly new research area and it is important that the architecture community is able to iterate quickly on architectural exploration. Although there are a number of commercial Deep Neural Network(DNN) accelerators available on the market and a plethora of creative ML architectures have been proposed in Academia, there exist only a few end-to-end DNN accelerators implementations which academics can readily study and use to inform future DNN accelerator developments.

A number of tools have recently surfaced to help address this need. Some of these include advanced RTL design tools and compilers that consume ML framework output and emit instructions for custom accelerators. However, creating an end-to-end accelerator is still quite difficult. There are a number of hurdles to overcome including writing drivers, achieving high transfer speeds between host and accelerator, modifying compilers to support custom hardware, choosing a the correct bus to support connecting the accelerator fabric to the chosen memory system, and even choosing the right RTL.

This thesis documents the process of building an end-to-end accelerator complete with a custom compiler in the hopes that highlighting the most difficult parts of creating complete accelerator systems informs the techniques used by future architects and system designers.

CHAPTER 1
INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS

All the code needed to reproduce the results in this thesis can be found at the following links:

1. https://github.com/BracketMaster/reproduce-thesis
2. https://github.com/BracketMaster/maeri

1.1 Deep Neural Networks

The development of Deep Neural Networks(DNNs) over the past ten years has enabled incredible advancements in image classification, speech recognition, natural language processing, and text translation.

DNNs constitute a subset of Machine Learning(ML) which is in turn a subset of AI. The rough chronological relationship is depicted in Figure Figure 1.1.

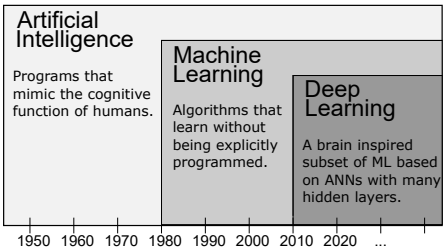

Figure 1.1: Chronological relationships between different AI fields : from [2]

A Neural Network is composed of layers. A layer in a Neural Network contains Artificial Neurons. These Artificial Neurons are inspired by their biological counterpart but are notably simpler. A Neuron can have multiple inputs and a single output as shown in Figure Figure 1.2.

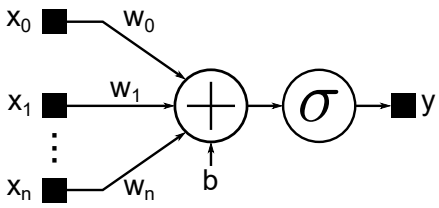

Figure 1.2: Graphical representation of a neuron : from [2]

A Neuron is a scalar valued function that operates on N inputs $(x_1, x_2, ..., x_n)$ and N weights $(w_1, w_2, ..., w_n)$ and returns a scalar valued y as described in EQ Equation 1.1.

$$y(\vec{x}) = \sigma(\sum_{n=0}^{N-1} \vec{x}[n]\vec{w}[n] + b) \tag{1.1}$$

σ in EQ Equation 1.1 represents an activation function. The activation function is typically non-linear. Without the activation function, the layers within a Neural network become redundant and can be flattened. This is because from the first layer, until the last, the Neural Network would be composed of purely linear transformations.

Common activation functions include Rectified Linear Unit(ReLU), Sigmoid, or Hyperbolic tangent.

For a Neural Network to be useful, it must be trained to produce the expected output over a typical range of inputs. Training a Neural Network requires:

- A way to score the Network's error given a certain input.
- A way to update weights such that iteratively reduces Network error.

Neural Network error scoring functions are commonly called loss functions. Since Neurons tend to be composed of differentiable functions, it is possible to compute the gradient of the loss function with respect to any weight within the Neural Network. This gradient, $\frac{\partial L}{\partial w}$, can be used to update the weights. Under the assumption that

the loss function is convex and has a minimum somewhere within the input space, the weights in the Neural Network can be updated such that the loss function is minimized each scoring-update iteration.

In DNN literature, scoring a network is referred to as inference, and using the gradient to update weights is usually referred to as back-propagation.

Consider two adjacent layers, l_1 and l_2 in a Neural Network. If each Neuron in l_2 uses all the outputs of the Neurons in layer l_1 as its inputs, layer 2 is said to be fully connected to layer 1.

Fortunately, for many practical domain-specific applications of Neural Networks, it is sufficient to use layers in which the Neurons within only "see" a subset of all Neuron outputs of the previous layer.

Convolutional Neural Networks

Image Classification is one such domain. The core operator in image classification Neural Networks is the convolution operator. Such networks are typically called Convolutional Neural Networks(CNN). Most layers within CNNs typically perform convolutions.

Only 2d Convolutions are treated here since the accelerator implemented in this thesis supports only 2d Convolutions. A 2d convolution operates on two inputs $\boldsymbol{X}$ and $\boldsymbol{W}$ to produce and output $\boldsymbol{O}$ as detailed in EQ Equation 1.2.

$$O[c_o, h_o, w_o] = \sum_{c_i=0}^{C_i-1} \sum_{h_k=0}^{H_k-1} \sum_{w_k=0}^{W_k-1} \boldsymbol{W}[c_i, c_o, h_k, w_k] \boldsymbol{X}[c_i, Sh_o + h_k, Sw_o + w_k] + \boldsymbol{b}[c_o] \quad (1.2)$$

Figure 1.3 illustrates a convolution between a 3x3 input and a 2x2 weight. Both the input and weights can have multiple channels. In Figure 1.3, the channels are represent by different shades of the same color.

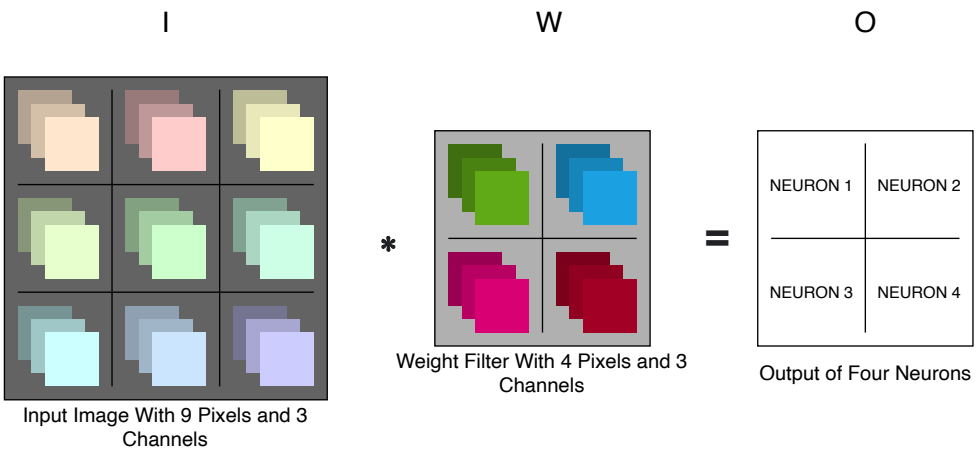

Figure 1.3: Convolution between input and weight producing output.

A 2d convolution can be thought of as a sliding summed product. The resulting output in Figure 1.3 is composed of four values. Each one of these values is the output of a neuron.

In a 2d convolution, the weight matrix does not slide channel-wise. Both the input and the weight matrix must have same channel dimensions. Also, during 2d convolution, these channel dimensions are aligned. Channels often correspond to the RGB components of an image.

Sliding the weight matrix left to right, top to bottom within the input matrix produces the output matrix. To slide the weight matrix within the input matrix, all dimensions of the weight matrix must not be greater than those of the input matrix. Common weight matrix dimensions in the top CNNs don't exceed 7x7, so this constraint is never violated.

The entries of the output matrix can be treated as neurons. Figure 1.5 shows how each neuron would operate on the weight and input matrices.

1.1.1 Example CNN

Since the bulk of this thesis documents the process of translating a Neural Network model into an executable that then gets executed by an FPGA accelerator, it is only

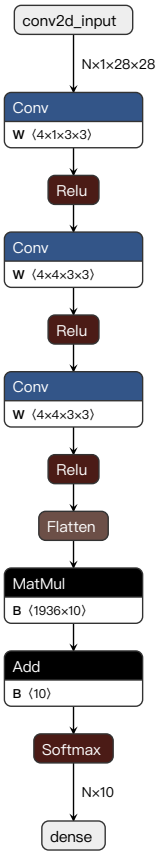

Figure 1.4: Computation graph for an MNIST classifier.

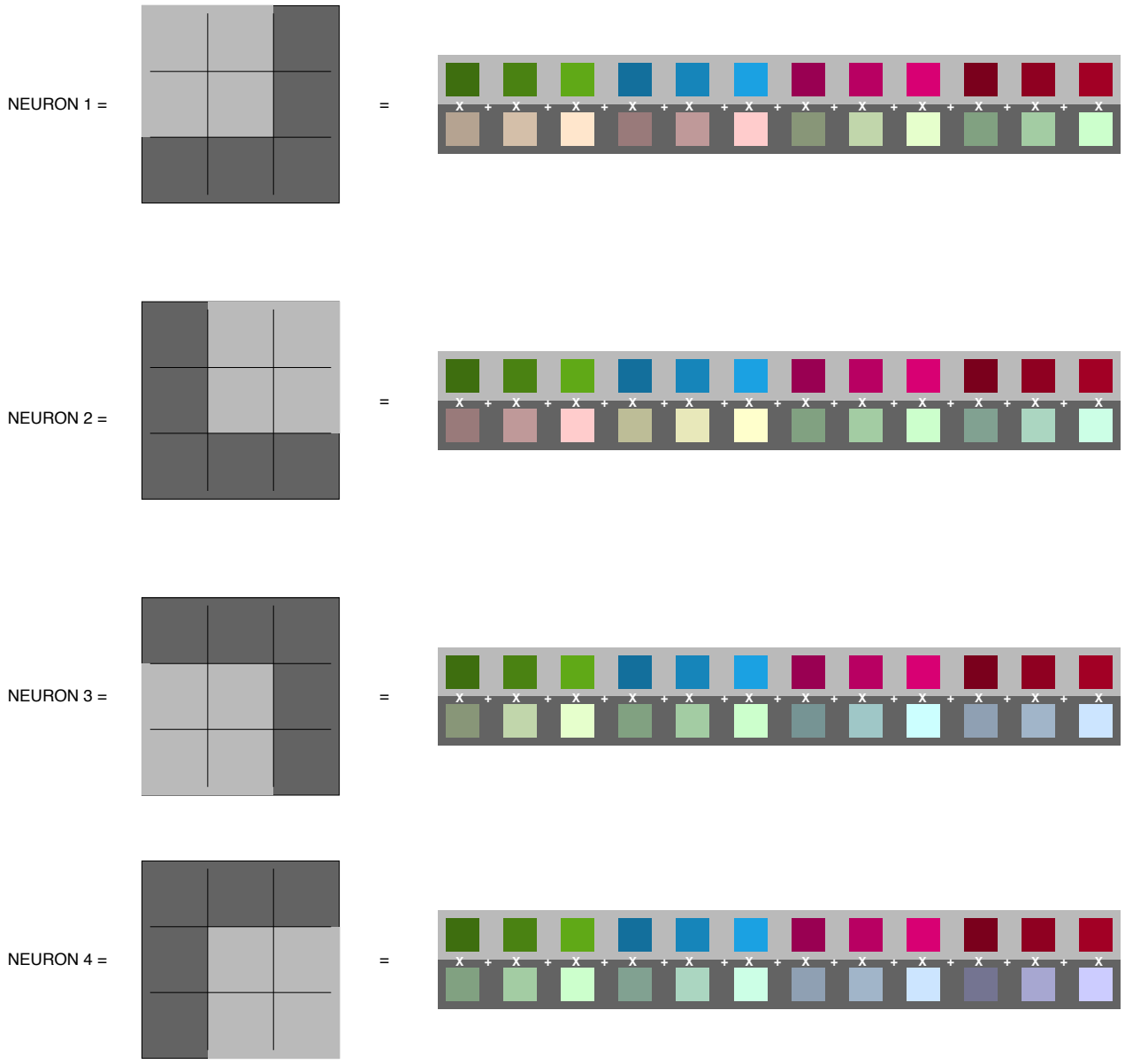

Figure 1.5: Computations from the perspective of a Neuron.

apt to start by looking at a simple example Neural Network model.

MNIST is a dataset consisting of 60,000 handwritten digits. An MNIST classifier is the hello world of CNNs due to its simplicity. A CNN accelerator designed to support MNIST classification can be extended to support deeper and more complex CNNS in a fairly straightforward fashion. For this reason, MNIST is deployed on the accelerator designed in this thesis as a baseline.

A graphical depiction of the simple MNIST classification network referenced through-

Table 1.1: Computational breakdown of Thesis Classifier by layer

Layer	Mults	Adds
CONV1	2713	2732
CONV2	9225	9244
CONV3	7753	7772
MATMUL1	19360	19350
SOFTMAX	NA	NA
TOTALS	39051	39098

out this thesis is provided in Figure 1.4. From here onwards, to avoid repetition, we refer to the classifier in Figure 1.4 as thesis-classifier. Note that thesis classifier has two convolutional layers and one fully connected layer also known as matmul. Also note that softmax can be ignore if it is the last layer during inference.

Thesis-classifier also has 20,147 trainable parameters. Table 1.2 tabulates some seminal CNNs which include some modern state of the art CNNs, all which have parameters in the millions. Thesis classifier has an accuracy of 99%, which is about the same as LeNet. LeNet however has 12 times as many parameters as thesis classifier. Table 1.1 shows the computational breakdown by layer for thesis classifier, which has a total of roughly 80,000 computations. Some higher accuracy models are tabulated in Table 1.2 and with parameters in the millions. It is thus not unreasonable to expect such models would have computations in the billions.

1.2 The Need for CNN Accelerators

CNNs are able to deliver results rivaling and sometimes surpassing humans when given image classification tasks. As mentioned in the previous section, even the simplest of networks are extremely computationally expensive. Current research suggests that custom hardware can enable real time classifications within a low power budget.

Although accelerators are still a fairly recent area of research, a number of accelerators have been deployed in the marketplace as detailed in table Table 1.3. Such

Table 1.2: List of Selected Seminal Convolutional Neural Networks: adapted from [2]

Model	Year	Contribution	No. Param	Depth	ImageNet Accuracy
LeNet[3]	1998	First popular CNN	60 k	5	-
AlexNet[4]	2012	First CNN to win ILSVRC ReLUintroduction	60 M	8	79.06
VGG16[5]	2014	Smaller kernel sizes	138 M	16	90.37
GoogLeNet[6]	2015	Inception block	4 M	22	87.52
Inception v3[7]	2015	Factorized Convolutions	24 M	48	93.59
Inception v4[8]	2016	Simplified Inception Blocks	43M	77	95.30
ResNet 50[9]	2016	Skip Connections	26 M	50	92.93
ResNet 152[9]	2016	Residual Learning	60 M	152	93.98
Xception[10]	2017	Depthwise and Pointwise Convolutions	23 M	38	94.50
ResNetXt 101_64x4d[11]	2017	Grouped Convolution	83 M	101	94.70
DenseNet161[12]	2017	Regular Structure and Information flow across layers	28 M	161	93.60
SeNet154[13]	2018	Explot dependencies between feature maps	115 M	154	95.53
NasNet-A[14]	2018	Neural Architecture Search Transfer Learning	89 M	29	96.16

rapid marketplace adoption suggests that Neural Network accelerators are more than a niche academic interest.

1.3 Contribution

MAERI[1] is a DNN accelerator design methodology that the accelerator presented in this thesis conforms to. The key contributions of this thesis and the accompanying source code are as follows:

- Documentation and insight into the ML accelerator creation process.
- Functional DNN Accelerator FPGA implementation with 100% FOSS flow including FOSS synthesizer, place and router, packer, and programmer.

Table 1.3: NN accelerators in Industry

Accelerator	Consumer Facing?	Supports Training?
GraphCore IPU	Yes	Yes
Apple DNNS	Yes	Yes
Cerebras CS-1	No	Yes
Google TPU	Partially	Yes
Edge TPU	Yes	No
Intel Compute Stick	Yes	No
Amazon Inferentia	No	Unknown
Microsoft Brainwave	Partially	Yes

- Functional DNN assembler that can be extended to a full DNN compiler for targeting MAERI. extend.

The flow for the toolchain involves three steps as shown in Figure 1.6.

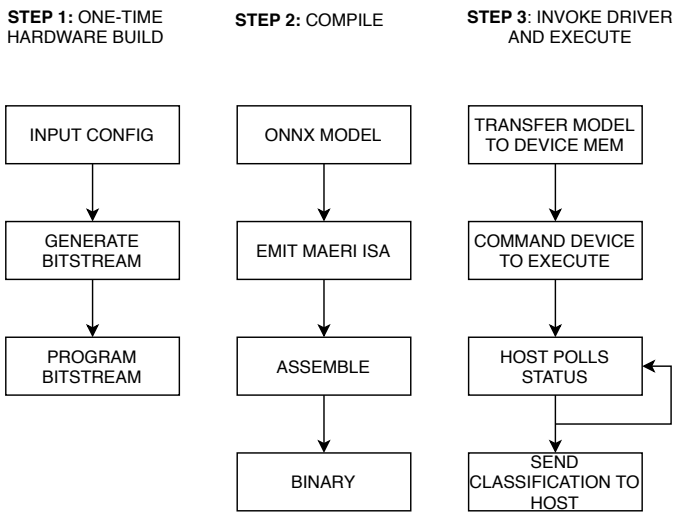

Figure 1.6: 10 mile high depiction of the toolchain flow specific to this thesis.

1.4 Structure of Thesis

This thesis is structured as follows: Chapter I provides an overview of DNNs and sets up the motivation for DNN accelerators. Chapter II surveys the landscape of DNN accelerators and compilers targeting these accelerators. Chapter III methodically introduces various design points and factors to be considered when designing a DNN

accelerator. Chapter IV reviews similar considerations when building a compiler targetting DNN accelerator hardware. Chapter V compares the runtime performance of the accelerator implemented in this thesis to various other accelerators. Chapter VI discusses future directions research directions for the accelerator implemented in this thesis.

CHAPTER 2
BRIEF SURVEY OF NEURAL NETWORK ACCELERATORS AND METHODOLOGIES

2.1 Taxonomy of NN Accelerators

The computer architecture community has had about 6 or so years to explore various NN architectures and has converged on a number of attributes common to most NN accelerators.

Common to all NN accelerators are processing elements(PEs). PEs usually contain adders, multipliers, and sometimes a tiny bit of memory.

2.1.1 Dataflow

One of the first problems computer architects encountered when exploring accelerator design was the high cost to access large memories. There are a couple ways to avoid paying high memory access penalties, namely, accessing the memory less frequently and or developing a memory hierarchy.

Dataflow within the computer architecture community refers to the rules that govern memory access patterns for a particular computation. For example, one could design an accelerator that minimizes the number of weight fetches from memory. One way to do this is to design the accelerator such that the weights in the NN can be pinned to the memories in the array of PEs at runtime for the duration that the weights are used. Such an accelerator would be classified as weight stationary.

An output stationary accelerator functions such that by the end of runtime for a particular computation, the final results or output of the computation resides within the array of PE memories. An accelerator using an output stationary systolic array

would be one such example.

Another notable dataflow scheme is row-stationary. Such accelerators usually involve pinning rows to local memories within the array of PEs for the duration of a computation.

2.1.2 Sparsity

It is not uncommon for NN weights and inputs to have a substantial number of zero valued entries due to RELu activations, padded inputs, or network pruning. Since anything times zero is zero, if the accelerator knows ahead of time which entires or zero, it can avoid fetching fetching those weights. If zero weights will never be fetched, then it would be advantageous to not even store them in memory.

There are a number of compression schemes that avoid storing zeroes in memory. Some common ones include Compressed Sparse Row(CSR) and Compressed Sparse Column(CSC).

CSR is a fairly straightforward scheme. Consider the matrix below.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

Its CSR representation would be:

```
V         = [ 5 8 3 6 ]
COL_INDEX = [ 0 1 2 1 ]
ROW_INDEX = [ 0 0 2 3 4]
```

The `V` list contains all the non-zero elements in the matrix. The `COL_INDEX` list contains the columns of the respective non-zero entires. The `ROW_INDEX` list is a bit more involved. The entries in the `ROW_INDEX` list are indices for entries in the `V` list.

Table 2.1: Selected accelerators supporting sparsity.

Accelerator	Ref	Contribution	Target	Year
Cnvlutin	[15]	CSR for activations	ASIC	2016
Cambricon-x	[16]	CIS for weights	ASIC	2016
SCNN	[17]	CSC for weights and activations	ASIC	2017
Sparten	[18]	Improvement of SCNN	ASIC	2019
EIE	[19]	CSC for weights, zero-skip for activations	ASIC	2016
NullHop	[20]	CSC for weights, zero-skip for activations	FPGA	2018
ZeNA	[21]	Zero-skip of weights and activations	ASIC	2017
SqueezeFlow	[22]	RLX for weights, concise convolution rules	ASIC	2019
Eyeriss v2	[23]	CSC for weights and activations, data are kept compressed during computation	ASIC	2019
UCNN	[24]	Generalizes sparsity to non-null weights	ASIC	2018

`ROW_INDEX[i]` and `ROW_INDEX[i + 1]` represent the starting and ending index of the i^{th} row.

Another compression scheme is run-length encoding which is quite simple. Run-length encoding involves specifying the dimension of the data, and then creating a list filled primarily with non-zero values. Any time that list has a zero entry, the following entry must be non-zero and is interpreted as specifying the repeat count of the zero.

An accelerator with sparsity capabilities does incur a bit of a overhead for the hardware needed to transform data from its sparse representation to its actual representation.

Table 2.1 lists some selected accelerators supporting sparsity.

2.1.3 Memory Hierarchies

Modern Neural Networks can have millions of parameters and therefore must be stored in large memories. Access times to large memories have high energy and latency costs. One was to combat this is by introducing intermediate smaller memories into the system. Since the exact runtime computation graph for CNNs is known at compile time, simple input, weight, and activation buffers are often used in accelerators instead

of caches.

Some exotic architectures have been proposed such as embedding an accelerator in a camera sensor avoiding the need for DRAM entirely. Logic-in-memory architectures have also been proposed.

2.2 Related Works in DNN Accelerators

The end-to-end implementation of an accelerator presented in this thesis is in no way the first attempt at an open end-to-end accelerator. There are two open end-to-end accelerator implementations worth mentioning, namely: NVDLA and VTA.

There are also two plug-and-play proprietary accelerators on the market the author knows of at the time of this writing, namely Google's edge-TPU and Intel's compute stick.

NVDLA

NVIDIA Deep Learning Accelerator(NVDLA)[25] is a complete open source NN inference accelerator solution designed by NVIDIA. The accelerator itself is highly modular and parametrizable and includes both a SystemC implementation for performance modeling as well as an RTL implementation. The NVDLA accelerator can be composed of multiple NVDLA cores. And NVLDA core can contain a convolution core, a single data processor which performs lookup for activation functions, a planar data processor for averaging and pooling, a channel data processor for channel wide averaging and normalization, and a memory and data reshape engine. These different components can be instantiated independently to different configurations depending on the constraints of the target application. For example, the convolutional core mac array size can be adjusted from 64 to 4096.

NVDLA also offers memory flexibility with up to two AXI busses. One of these busses can be connected to a large system memory while the other can be connected

to an external device specific memory solely for the NVDLA accelerator.

NVDLA comes with complete software stack support. The NVDLA compiler can consume a pre-compiled Caffe model and lower it into primitives that the NVDLA hardware supports. In doing this, the compiler must be able to reason about the particular hardware instantiation attributes such as number of PEs, or whether or not the convolution cores offer winograd convolution acceleration. These primitives get packed into a format called an NVDLA loadable. Dependencies between operations can also be specified in an NVDLA loadable. The NVDLA core requires a host controller for operation. The NVDLA driver stack is designed to be elastic, that is, a host CPU could exercise fine grained control over the accelerator, or the CPU could exercise coarse grained control, delegating fine grained control to a microcontroller that's part of the accelerator. The stack that NVDLA comes with supports submitting an NVDLA loadable to a user mode Linux kernel driver. A kernel mode driver then selects between various available tasks from the user mode drivers and batches them into coarse grained schedules that are submitted to a microcontroller on-board the NVLDA accelerator.

VTA

Versatile Tensor Accelerator(VTA) is another accelerator that is part of the end-to-end accelerator stack TVM[26] from University of Washington. The notable and differentiating feature of VTA is that an accelerator configuration can generated for the particular Neural Network model to be evaluated. The core of VTA is notably simpler than that of NVDLA with only a GEMM and ALU core, and no reshape or pooling engine. These functionalities can be handled by the ALU core to some extent. NVDLA is designed to support multiple NVLDA core instantiations within the same accelerator whilst VTA currently only supports one core.

VTA utilizes the TVM compiler as its interface to higher level machine learning

frameworks. The TVM framework transforms a high level NN model into TVM IR which is then consumed by a VTA JIT compiler along with the particular VTA configuration and produces a binary. Unlike the NVDLA loadable, the VTA binary is simply a microkernel that only contains a few operations. This is because the VTA lacks a proper controller and requires more intervention from the host. The flow for VTA involves consuming a model in TVM IR format and repeatedly jitting it into microkernels until the entire model has been executed.

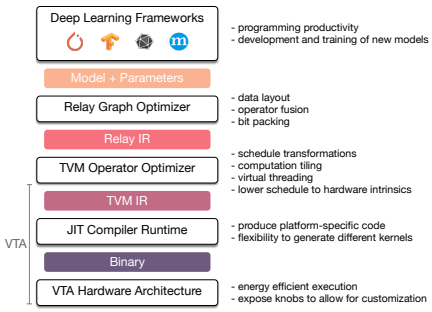

Figure 2.1: VTA stack

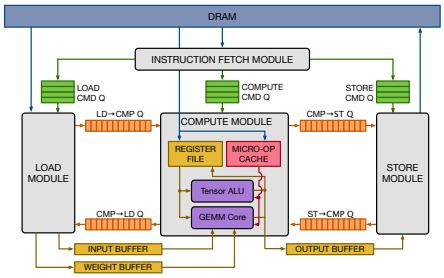

Figure 2.2: VTA hardware.

2.3 Related Works in DNN Compilers

Hardware accelerators often require custom compilers and DNN accelerators are no exception. There are two big reasons for this:

1. DNN accelerators ISA have domain specific primitive such as convolutions and matmuls.

2. Operations such as convolutions as shown in Listing 2.1 are hard for traditional compilers to reason about and optimize as they are expressed with the wrong level of granularity.

Listing 2.1: Convolution loop-nest example.

```
for (n=0;n<N;n++) {
    for (m=0;m<M;m++) {
        for (c=0;c<C;c++) {
            for (i=0; i<H; i++) {
                for (j=0;j<H;j++) {
                    for (rx=0;rx<R;rx++) {
                        for (ry=0;ry<R;ry++) {
                            O[n][m][i][j] = ... }}}}}}
```

There are a few notable compilers that have been developed specifically for DNN accelerator hardware, namely: TVM, GLOW[27], MLIR-TF[28], and XLA.

XLA is primarily built to support Google's TPUs and little support is provided for other custom architectures. Both TVM and GLOW allow for custom backends to support lowering to custom accelerator hardware. MLIR is a generic compiler building framework that supports domain specific dialects. MLIR has a dialect for TensorFlow called MLIR-TF that can also be extended to lower to custom hardware accelerator targets.

The author found that the GLOW compiler was easiest to build but somewhat difficult to extend give GLOW's lack of soft-documentation. TVM was easiest to extend but difficult to build consistently. MLIR was impossible to build - the author was able to create a functional build environment - also - as of the time of writing, the build instructions for the MLIR tutorials were out of data and did not track the MLIR master branch. The author did not touch XLA as is was only built to support

Google's TPU.

2.4 Target DNN Accelerator: MAERI

MAERI[1] is an accelerator design methodology proposed by the Synergy Lab at Georgia Tech. The accelerator is designed for computations common to RNNs and CNNs. The accelerator principles are actually quite simple. The computation core is in essence an adder reduction tree who's leaves are multipliers.

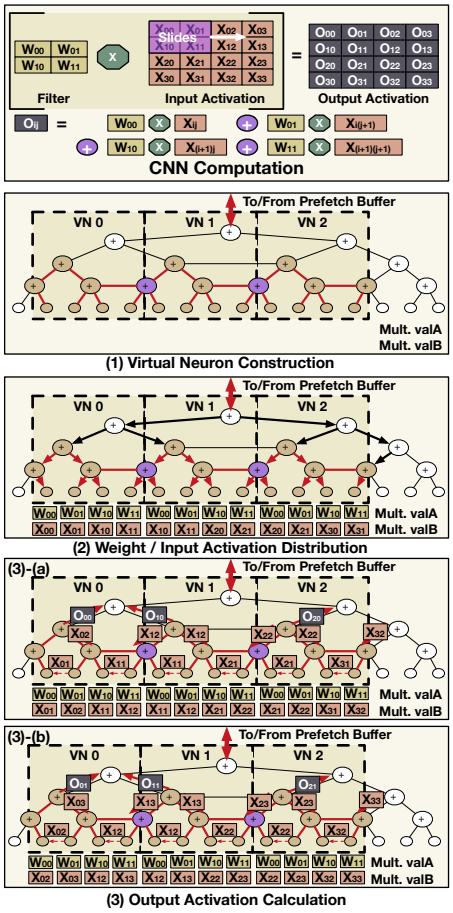

Figure 2.3: Convolving W and X with a MAERI instantiation to produce O : from[1]

The accelerator design in this thesis is inspired by MAERI. The author chose to design a MAERI-like accelerator in order to continue and complete an effort that began within the Georgia Tech Synergy lab.

Table 2.2: Some common accelerators on the market and MAERI.

Accelerator	Substrate	Compiler	Comm Link	Plug and Play
Edge TPU	ASIC	edgetpu_compiler	libusb	Yes
Intel Compute Stick	ASIC	OpenVino	libusb	Yes
MAERI	FPGA	MAERI Assembler	libusb	Yes
VTA	FPGA	VTA JIT	unknown	No
NVDLA	Designer Define	NVDLA	Designer Defined	No

Table 2.2 provides a quick comparison of some high level features of the accelerators mentioned in this introduction.

CHAPTER 3
DNN ACCELERATOR DESIGN

3.1 Choosing an RTL

The author used Bluespec, Verilog, Chisel, and Migen to build incomplete versions of the hardware portion of the MAERI DNN accelerator before converging on using nMigen as the HDL of choice. In the search of the right HDL, the author identified the criteria in Table 3.1.

1. and 2. from Table 3.1 make for a sufficient RTL. Verilog fails to satisfy 1. and 2. `reg` in Verilog is functionally identical to `wire`. Delays in Verilog are introduced through blocking assignments in Verilog procedural blocks. In addition, verilog makes for a miserable general programming experience which led the author to abandon verilog generate statements for creating the binary adder tree portion of the MAERI core.

Although bluespec satisfies all properties listed in Table 3.1 except for first class support for formal verification and low learning curve. Bluespec's high learning curve and poor readability made it difficult to delegate and handoff design subtasks to peers. Seeing that bluespec was unsuitable for collaboration, the author abandoned it.

Chisel and nMigen satisfy 1. and 2. Chisel accomplishes delays in a domain with `RegNext` while nMigen has the designer append statements to a synchronous domain list with `m.d.domain += statement` to achieve delays.

For the author, using Chisel presented two major challenges. Firstly, setting up the Chisel Environment was quite difficult and involved matching the correct versions of Scala, OpenJDK, and Chisel.

Table 3.1: Criteria for a Good RTL

	Features	VHDL	Verilog	BlueSpec	Chisel	nMigen
1	only two primitives: wires that function as direct connections and registers that introduce a propagation delay	No	No	Yes	Yes	Yes
2	support for general purpose programming	No	No	Yes	Yes	Yes
3	first class support for clocks	No	No	Yes	Yes	Yes
4	easy to write testbenches	No	No	Yes	No	Yes
5	low-latency simulation engines for rapid iteration on smaller modules	Simulator Dependent	Simulator Dependent	Yes	No	Yes
6	first-class support for formal-verification	No	No	Partial	No	Yes
8	easy to bringup build environment	Tool Dependent	Tool Dependent	Yes	No	Yes
9	expressible in an un-ambiguous RTL format	No	No	No	Yes	Yes
10	low learning curve	No	No	No	No	Yes

Secondly, although it is possible to use the Scala compiler manually and point it the location of various Scala library dependencies, chisel-lang recommends and only officially supports using SBT. SBT is quite slow and has a minimum build time of 5 seconds on the simplest of Scala programs.

Needless to say, rapid prototyping and design iteration was quite slow in Chisel. Chisel is a general purpose programming language, and as such, it was possible to express MAERI's binary adder tree without much difficulty, however, Chisel is written in Scala, which has a high learning curve, making Chisel also unsuitable for task handoff. Lastly, Chisel has no support for formal verification.

nMigen draws heavily on Migen for inspiration. The most distinguishing feature of nMigen is its close coupling to Yosys. Like Chisel which can compile down to an unambiguous FIRRTL, nMigen compiles down to RTLIL, both of which can be ingested by Yosys for synthesis. The most compelling property of nMigen is that

its RTL-core along with its simulation engine, are written entirely in Python. This affords nMigen the following benefits:

1. nMigen is highly accessible thanks to Python being the most popular programming language.

2. nMigen is very readable

3. A purely Python implementation allows nMigen to be easily installed on almost any system supporting the pip Python package manager

4. The simulation engine can run nearly instantaneously allowing for rapid iteration on small designs.

5. The hardware designer has access to many Python packages allowing for powerful and flexible workflows such as using SciPy to design an FIR filter to a particular spec, and then invoking nMigen to compile and synthesize hardware.

nMigen also has a few more features the author appreciates which are technically not symptoms of nMigen being a Python Domain-Specific-Language(DSL).

1. out of box support for the most common HDL patterns such as state machines, elastic buffers, and SRAMs

2. nMigen provides a small set of primitives allowing the user to extend and derive the primitives in the fashion most suited to his or her workflow

3. first class FPGA support rendering rapid prototyping very easy

The Python simulator included with nMigen is called PySim. Although PySim has very low overhead and starts simulating instantly, it is implemented in Python which prevents it from achieving high speeds. nMigen can be compiled directly into C using Yosys for a substantial increase in simulation speeds. nMigen can also be compiled

into Verilog with Yosys, which can then be simulated with Verilog simulators such as Verilator.

3.1.1 Could Simplicity be Best?

At the end of the day, nMigen is by far the simplest and most primitive HDL the author has encountered. Primitive in the sense its core only contains a few components off of which everything else is built. Yet, the author has found nMigen to be the most productive HDL he has used.

Could it be that simplicity is best? Each component in nMigen only does one thing, but does it really well. nMigen's author, who goes by the name of Whitequark has had much experience creating tooling around Verilog, and has had ample time to learn and improve on Verilog's shortcomings.

This isn't to say that nMigen is perfect. Its testbench tooling would benefit from the fork and join amenities that Chisel provides for example.

3.1.2 Some Issues With Verilog

Upon request of the author, Whitequark took some time to enumerate some of Verilog's shortcomings, which are listed below.

1. Most of its constructs are not synthesizable.
2. A composition of synthesizable constructs may not be synthesizable.
3. SystemVerilog makes no attempt to define which constructs are synthesizable. For example, `always_comb` and `always_sync` are 100% implementation defined and essentially non-portable.
4. *Improved* SystemVerilog features still have severe defects, e.g. `always_comb` is supposed to fix the problem of `always @` not triggering at time 0 (which caused a sim/synth mismatch), and it does that, but introduces the problem of missed

triggers in `always_comb begin a = b  b = c  end` (if `c` changes, `a` will end up wrong in simulation).

5. Simluation semantics is inherently and deliberately nondeterministic.

6. Even though Verilog coding styles that avoid problems with e.g. blocking/non-blocking assignments exist, they have many edge cases and cannot be applied mechanically. E.g. clock gating circuits must use blocking assignment in an always @`(posedge)` block.

7. Basic arithmetics has extremely surprising behavior, in particular around integer promotion. `Signed` + `Unsigned` yields `unsigned`. Also, width of an expression depends not only on the expression but on context in which it is used.

8. SystemVerilog is a massive standard (which essentially no vendor implements in full), and it offers no way to subset it to be able to claim compliance meaningfully.

9. Using memories in a portable way requires relying on inference, which cannot happen either on syntax level (or you would restrict coding style too much), or on netlist level (or you would miscompile some inputs). E.g. a synchronous, transparent read port is expressed using an idiom that combines an asynchronous read port with registered address. However these have different semantics. If you actually need an asynchronous read port, but your netlist happens to drive it with a register (which may be on a completely different level of hierarchy) then you will get a miscompilation.

10. Conflation of 'x meaning "timing violation", 'x meaning "uninitialized register/memory" and 'x meaning "this value left open for optimization" means that

perfectly correct (even formally verified) modules can be miscompiled (and produce seemingly impossible results like `a && !a`) if they are fed a 'x through the ports. (There is no way to avoid this with commercial synthesizers).

11. "Structural Verilog" doesn't exist but many tools claim to generate or consume it. They are not compatible with each other.

12. `generate` is both highly complicated to implement (which means it is often not supported well), and restricted in the amount of logic it can produce, meaning people resort to preprocessing with perl anyway.

13. Even though lots of tools generate Verilog, there is no standard way to serialize location info beyond crude preprocessor directives, meaning all that generated Verilog is extremely hard to debug.

14. API for interacting with the outside world is extremely painful: you have a choice between crude and non-portable stdio bindings, and DPI-C, which is unsafe and crashes a lot.

15. The standard waveform dump format is extremely limited. E.g. no way to determine the sign of a signal, or symbolize enums.

16. No standard library, or portable way to mark clock domain crossing.

3.1.3 FIR filter in Chisel and nMigen

For the benefit of the reader, a parameterized FIR filter listing is provided below in both Chisel and nMigen.

nMigen FIR filter

```
class DSP_chain(Elaboratable):
    def __init__ (self, weights,WIDTH):
```

```
        self.weights = weights
        self.WIDTH = WIDTH
        self.sig_in = Signal(signed(WIDTH))
        self.sig_out = Signal(signed(WIDTH))

    def elaborate(self, platform):
        m = Module()

        m.d.comb += reg_array[-1].eq(self.sig_in)
        reg_array = Array(Signal(self.WIDTH, name=f"delay{i}") \
                for i in self.weights)
        signal_pairs = zip(reg_array[:-1], reg_array[1:])
        for next_val, prev_val in signal_pairs:
            m.d.sync += next_val.eq(prev_val)

        weighted_sum = sum([el*weight \
                for el,weight in zip(reg_array, self.weights)])
        m.d.comb += self.sig_out.eq(weighted_sum)
        return m
```

Chisel FIR filter

```
// Generalized FIR filter parameterized by the convolution coefficients
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })
```

```
  // Create the serial-in, parallel-out shift register
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

  // Sum up the products
  io.out := products.reduce(_ + _)
}
```

3.1.4 FPGA Programming in nMigen

nMigen has support for FPGA platforms and abstracts away the mundane details involved in invoking specific FPGA tools and creating pin mapping files. This abstraction is well designed such that it is quite easy for the user to extend nMigen board support or repair any bugs. The user can grab request an FPGA pin and treat it as a signal that can be assigned to any domain in nMigen. This functionality can be extended from the simple use cases of blinking an LED to controlling an SRAM. A listing below is provided demonstrating the simplicity of prototyping on an FPGA. It is even possible to develop rapid-FPGA prototypes with nMigen using completely FOSS flows. Such a flow might involve:

1. Developing HDL in nMigen.
2. Requesting the Relevant FPGA pins in nMigen as signals enabling the designer to interface with the outside world.

3. Compile nMigen down to RTLIL for Yosys to ingest

4. Yosys emitting a LUT netlist NextPNR to ingest

5. NextPNR emitting a tile and routing matrix configuration for icepack to ingest.

6. icePack emitting a bitstream for iceProg to ingest

7. iceProg programming the FPGA

Since Python can call tools in its own subshells, nMigen can actually abstract away the entire process from steps 3 to 7 listed above in a simple `platform.build()` command. The following listing provides a simple FPGA blinky example using the exact flow from the list above.

```
from nmigen import *
from nmigen_boards.ulx3s import ULX3S_85F_Platform


class Blinky(Elaboratable):
    def elaborate(self, platform):
        user_led = platform.request("led", 0)
        counter  = Signal(23)
        m = Module()
        m.d.sync += counter.eq(counter + 1)
        m.d.comb += user_led.o.eq(counter[-1])
        return m


if __name__ == "__main__":
    platform = ULX3S_85F_Platform()
    platform.build(Blinky(), do_program=False)
```

3.1.5 Testbenches in nMigen

Writing testbenches in nMigen is also quite simple. Find below an example testbench for the nMigen FIR filter presented earlier.

nMigen FIR Filter Testbench

```
def testbench():
    signal_in = [1,1,1,1,1,0]
    signal_out = []

    for value in signal_in:
        yield dut.sig_in.eq(value)
        signal_out += [(yield dut.sig_out)]
        yield

    for cycle in range(5):
        signal_out += [(yield dut.sig_out)]
        yield

    print(signal_out)

dut = DSP_chain(WIDTH=8, weights=[1,1,1,1])
sim = Simulator(dut)
sim.add_sync_process(testbench)
sim.add_clock(1/(1e9))

with sim.write_vcd(f"test.vcd"):
sim.run()
```

3.2 Choosing the Hardware Substrate

The ideal compute substrate for for a DNN accelerator would be an ASIC. The author chose to prototype the MAERI accelerator on an FPGA and leaves ASIC implementation to future graduate students.

The ULX3s FPGA was chosen for the following reasons:

1. It supports the FOSS place and router NextPNR and the FOSS synthesizer Yosys. The author finds these tools the easiest of their kind to use. They are also the only FPGA tools that run natively on MacOS, the authors compute platform of choice.

2. It sports a 32MiB SDRAM. Writing SDRAM controllers is notably easier than writing DDR3 controllers.

3. It's PLL supports 48MHz, allowing the author to use the FPGA fabric as a USB PHY.

3.3 Choosing the Right Host - Device Link

It is important to get as much of the DNN model in a memory as close to the accelerator as possible. In some cases, this might involve having the accelerator use DMA to move data from a higher level memory to a memory close to the FPGA.

In the implementation presented in this thesis, the FPGA accelerator cannot access the host memory directly. Thus, the model must be transferred to the FPGA over a link. The author had access to a few options, namely, UART over FTDI, 125MBps RGMII Ethernet, PCIe Gen 3 with 2 lanes, and USB.

The FPGA onboard the ULX3s is a Lattice ECP5 with 85,000 LUTs and 2-duals 4-channel SERDES. It is possible to configure these SERDES into a 2.5G 2-lane PCIe link. Doing so however requires purchasing black box IP provided by Lattice

which only works with the Lattice clarity verilog compiler, thus forgoing the amenities provided by NextPNR and Yosys. On the software side, writing PCIe drivers isn't particularly hard. The author wished to avoid Lattice clarity verilog tooling, and thus did not pursue the PCIe link rout.

The next option available to the author was RGMII ethernet. A FOSS SOC builder by the name of LiteX provides a portable ethernet controller written in Migen that offers out-of-the-box support for ECP5 FPGAs. One of the requirements for the accelerator developed in this thesis support ease-of-use and plug and play ability. The author had difficulty writing drivers capable of correctly and consistently configuring the network and ethernet properties on an arbitrary computer, so the author abandoned using ethernet as a link.

At its highest speeds, an FTDI-UART is just too slow. This leave USB. The author settled on an open source device side USB controller written in nMigen by the name of Luna that currently supports up to USB 2.0 on devices that offer USB 2.0 PHYs. Luna comes with out-of-the-box support for USB 1.1 on the ULX3s FPGA. It is possible to achieve USB 2.0 speeds on the ULX3s with Luna if the proper USB 2.0 PHY is installed onto the ULX3s via PMOD.

Although the max transfer speed for the implementation presented in this thesis is 1.1MB/s, installing the right additional hardware can easily bypass this cosmetic limit achieving speeds of up to 480MBps without the any change to the RTL codebase or driver stack.

LibUSB enables a systems engineer to write high-performance platform agnostic USB drivers. Lastly, writing USB drivers is fairly easy. The driver before-hand must know the UUID of its target device. The driver can that ask the host OS if a device with such a UUID is currently plugged in to the system. Lastly, the driver addresses and dispatched arbitrary packets to the USB device at the specified UUID. Table 3.2 sums up the authors findings.

Table 3.2: Pros and Cons of Various Host - Device Links

Link	Driver Writing Difficulty	Plug-And-Play	Max-Speed	DMA Support	Interrupt Support
UART	LOW	YES	.5MBps	NO	NO
USB	LOW	YES	480MBps	YES	NO
PCIe	MODERATE	DEPENDS	2GBps	YES	YES
ETHERNET	MODERATE	No	125MBps	No	No

3.4 System Architecture

Figure 3.1 shows a high level view of the different hardware components of the DNN accelerator and how they all connect.

3.4.1 Choosing Clock Domains

There are three major hardware components in this DNN accelerator system, the physical communication link, the logic in the communication clock domain, and the logic in the compute clock domain.

The communication logic and the compute logic are in separate clock domains so that that the respective logics can be updated independently without affecting the timing properties of the other. For example, it might be possible to revise the compute logic and reduce its critical path, allowing the designer to legally increase the clock speed in the compute domain. The clock speed in the USB domain however should remain fixed at 12MHz as required by the USB 1.1 spec.

The designer however is certainly free to choose more than two clock domains. It is important however, to remember that more clock domains require more synchronization logic for crossing clock domains. Such synchronization logic introduces latency. One such synchronization primitive is an elastic buffer also know as an asynchronous buffer. nMigen has the added benefit of requiring the designer to explicitly specify all clock domains. This make communicating between clock domains quite simple. All

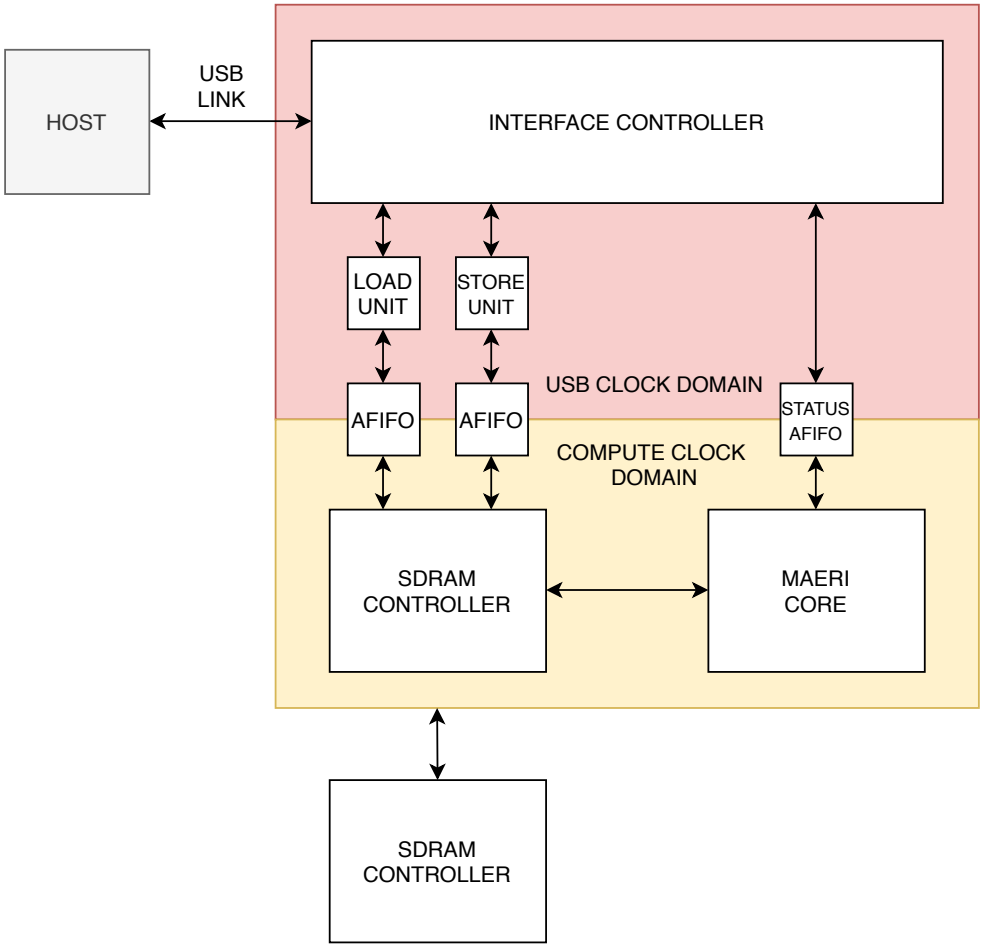

Figure 3.1: High level overview of thesis-DNN system architecture.

the designer has to do is instantiate an elastic buffer between modules in two different domains.

The author surmises that the following clock domain scheme could be the optimal one.

1. Clock domain for the communication logic
2. Clock domain for the memory controller
3. Clock domain for the MAERI compute core

Table 3.3: Packet protocol of the Interface Controller

Packet 1	Packet 2	Packet 3	Packet 4 .. N
“download”	address	length N	data from host
“upload”	address	length N	data to host
“r_status”	send status to host	—	—

3.4.2 Control

The host communicates with the accelerator by sending packets which are processed by the interface controller. The interface controller is quite simple and processes packets according to Table 3.3.

Due to the latency encountered when traversing clock domains, some logic is required to do burst transfers from the interface controller to the memory and vice-versa. This logic is captured in both the load and store unit.

3.4.3 Memory

SDRAM memory was chosen in this thesis implementation due to the ready availability of moderate performance open-source SDRAM controllers. Such a controller was used in this implementation. SDRAMs tend to be less dense than DDRAMs, thus, the autuhor recommends DDRAMs be used for DNN accelerator designs that demand more memory as the SDRAM presented here only provides 32MiB of memory. There are some downsides to DDR memories.

1. DDR memories require link training which is best done with a small CPU instead of dedicated logic. This ultimately results in a design that consumes extra resources.

2. As of the time of writing, LiteX Dram is the only open source DDR controller the author knows of. Setting up LiteX as a standalone DRAM controller is somewhat involved.

3. Choosing a black box vendor DDR controller requires abandoning the FOSS Yosys synthesizer and NextPNR place and router tools which the author prefers for reasons aforementioned.

3.4.4 MAERI Core

Skeleton

At the heart of the MAERI RTL design in nMigen is a Python class called Skeleton. The Skeleton class is composed of Node types of the Node class. Each node type has a left, right child, and an ID. One can easily create a balanced binary tree composed of Node types. The Skeleton class sets aside some of these nodes to be adders, and others multipliers.

The MAERI design core also includes some nMigen RTL classes, namely, AdderNode and MultNode. As per the MAERI design spec, AdderNode has three inputs, lhs_in, rhs_in, and forward_in. The MultNodes only have two inputs, feature_in and forward_in. The MultNodes also have internal weights that are configured by the config bus which will be discussed later.

The nMigen RTL class by the name of ReductionNetwork traverses Skeleton and connects the nMigen AdderNodes and MultNodes accordingly. It is also important to mention that each AdderNode and MultNode possesses an internal ID and a config bus. The config bus can address up to a total of 256 nodes. There can be more than one config bus. The amount of config busses to be instantiated is a parameter passed N to the nMigen ReductionNetwork RTL class upon instantiation. This class chunks all the AdderNodes an MultNodes together and creates N different groups. The class also create N config busses and hooks up each config bus to all the nodes in its group. A node only configures when it observes its address on the config line as well as the config_enable line high. The config bus can configure the AdderNode and MultNode states as well as set the weights within the MultNodes. One might ask about how

the compiler could possibly reason about all this. The compiler actually instantiates a build of the Skeleton class and queries the class about what nodes belong to which config groups so that it knows how to properly address configurations which are later assembled.

Ports

The MAERI core at its core is essentially a Load-Store architecture computer with a six instruction ISA and extremely large registers. These registers are in fact SRAMs. Upon instantiating ReductionNetwork(), the number of injection ports which is equal to the number of collection ports for the ReductionNetwork must be specified. These ports have SRAMs attached to them, which can also just be viewed as registers from the perspective of the ISA - more on the ISA later. The collection ports are somewhat different from the injection ports. As mentioned before, it is possible to specify the number of ports for a particular instantiation of the reduction network. A reduction network of depth D can have P ports where P is constrained on $1 \leq P \leq 2^{D-2}$. As mentioned before, nodes on the Skeleton each have an ID. This ID is actually assigned to a register internal to each register at the time the ReductionNetwork hardware is instantiated. A single collection port can select any of the outputs from nodes with ID n on the range $0 \leq n \leq \log_2 P - 2$.

This above written explanation of the collectors is somewhat dense. Figure 3.2 helps provide some clarity on exactly how the injectors, collectors, reduction network, and config groups all fit together for a reduction network of depth 5. Table 3.4 also enumerates the node members of various config groups for a depth of 5.

ISA

Beneath all of the abstraction, the MAERI core does essentially six things repeatedly:

1. Configure AdderNode and MultNode states

Table 3.4: Node Members by Config Bus for a Reduction Network of Depth 5.

Config Bus	Node Members
0	0, 4, 8, 12, 16, 20, 24, 28
1	1, 5, 9, 13, 17, 21, 25, 29
2	2, 6, 10, 14, 18, 22, 26, 30
3	3, 7, 11, 15, 19, 23, 27

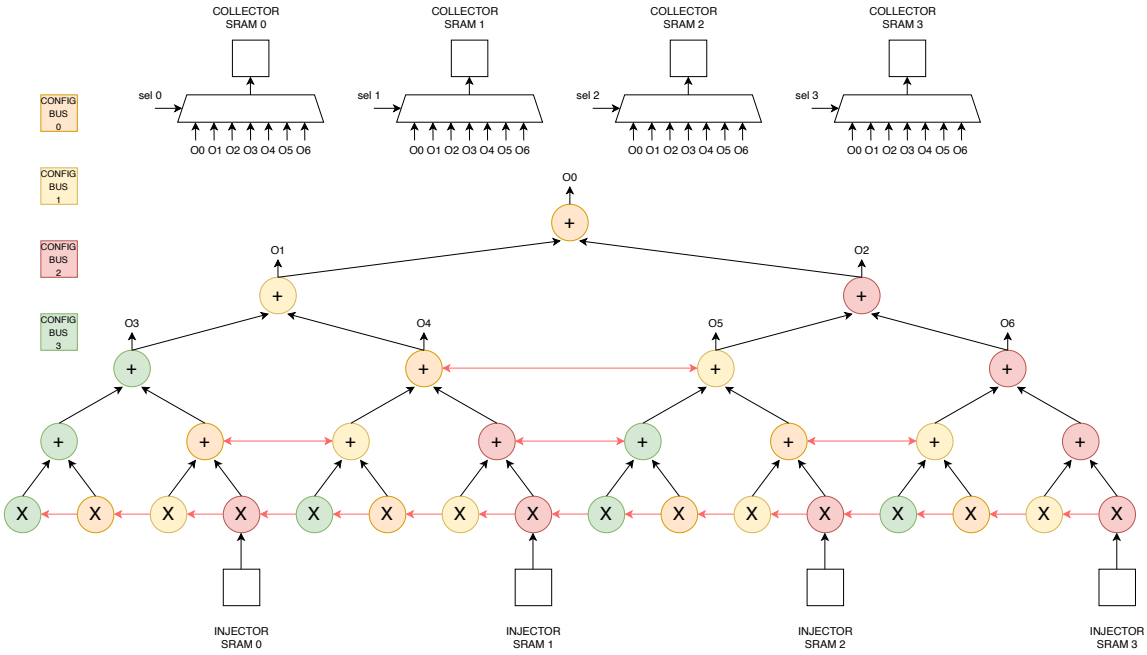

Figure 3.2: Architectural Diagram of the MAERI Core for a Reduction Network of depth 5.

2. Configure MultNode weights

3. Configure Collectors

4. Load injection SRAMs with features from device memory

5. Pump features over the ReductionNetwork for a specified number of cycles, the results percolate into the collection SRAMs

6. Store the resuts from the collection SRAMs back into the device memory

The resulting ISA is presented in Table 3.5. The opcodes are not presented in Table 3.5 because the opcodes are handled symbolically. The opcodes are hardened to

Table 3.5: Variable Width ISA.

BYTE 0	BYTE 1 - 4	BYTE 5	BYTE 6
CONFIGURE STATE	ADDRESS	N/A	N/A
CONFIGURE WEIGHTS	ADDRESS	N/A	N/A
CONFIGURE COLLECTORS	ADDRESS	N/A	N/A
CONFIGURE_RELUS	ADDRESS	N/A	N/A
LOAD FEATURES	ADDRESS	PORT	LENGTH
STORE FEATURES	ADDRESS	PORT	LENGTH
RUN	N/A	LENGTH	N/A

integers before the final stage of hardware synthesis. This is OK because the compiler can reference the Python hardware instantiation class before synthesis. Handling opcodes in this manner is advantageous because it avoids bugs that could be introduced when the need arises to update both the opcodes that are to be synthesized into hardware, and the opcodes consumed by the compiler.

CHAPTER 4
COMPILER

When designing a custom DNN accelerator, the author advises using the DNN compiler framework GLOW. The author could not figure out where to eject IR from the GLOW compiler in time for this thesis, so the author rolled a custom compiler that consumes ONNX IR for this thesis.

The author began implementing a compiler for this thesis which, as of the time of writing, is not currently operational. The compiler needs to be able to be presented with an operator and determine whether or not its possible to configure the MAERI Reduction Network for that operator, and then proceed to generate the configuration for that operator. Doing this, it turns out is very tricky and the author was unable to build a reliable configuration searchspace tool in time for the compiler. The author was also unable to get the compiler to reason about non-contiguous memory accesses. That is, is certain convolution or matmul may require loading an input that is current strided in the SDRAM, and loading such an input to be contiguous in SRAM buffers of the reduction network. This is extremely hard and the author was unable to figure out a reliable way to do this in time for the thesis.

Despite the Compiler's shortcomings, the design process and architecture of the compiler is discussed below in order to provide insight into the process of building a DNN compiler. Overall, custom compiler implemented for this thesis is quite primitive and will probably fail for models notably more complex than 3 layer MNIST models. It should be noted that the compiler implemented for this thesis can only consume models of datatype int8.

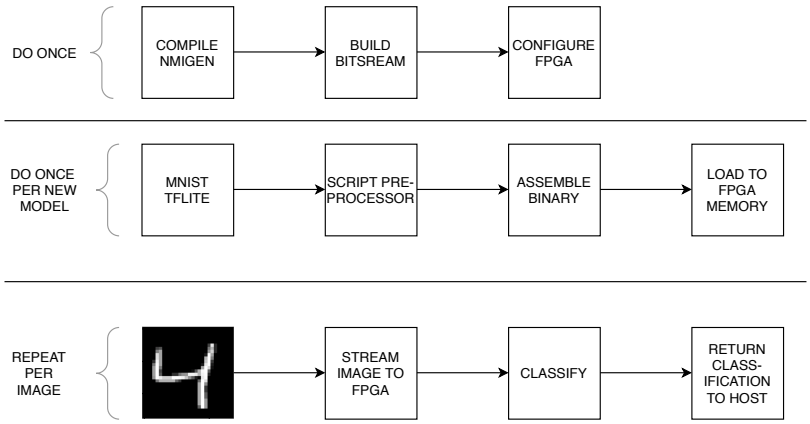

Figure 4.1: Workflow for using MAERI.

4.1 Canonical Operators

The compiler starts by transforming the ONNX model it recieves into a canonical format which consists of the following three operators:

1. SimpleConvs
2. Adds
3. Dot

Add and Dot-Product operators are self-explanatory. SimpleConvs refer to simple convolutions, where a single rectangular input is convolved with another rectangular input. It is quite common in CNNs for the CONV2D operator to allow for multiple channels and and multiple output filters. This means the compiler has to transform CONV2D operators into a graph that includes only SimpleConvs and Adds. Operators such as matmul must be transformed into multiple dot-products. This is because MAERI hardware can't actually perform matrix multiplications.

It should also be noted that the MAERI hardware has limits on the maximum size SimpleConv, Adds, or Dot-Product operations it can support. This means that after transforming all the input operators into one of the three supported operators, the

compiler must repeatedly split the operators until all of them are equal to or smaller in size than the hardware's maximum supported operator size.

The compiler must play special tricks to enable support for things such as batch-normalization or padding. To handle batch normalization, the compiler simply bakes the batch-normalization into the weights of the particular operator. Padding support is handled in a later lowering stage. To support padding, the compiler allocates a memory region filled with repeats of the value to pad with. The assembler would then take the SimpleConv that needs to be padded and generate LOAD_FEATURES instructions with references to the padding filled memory region as well as LOAD_FEATURES with references to the memory containing the actual SimpleConv operand features to be operated.

4.2 Compiling Memories

The compiler must be able to reason about two kinds of memories, variable memories and constant memories. Once the compiler transforms all the operators into valid operators supported by the hardware, it then forms a schedule out of these operators that is totally ordered. The compiler then passes three lists to the assembler, a constant memories list, a variable memories list, and a operators list.

4.3 Assembler

The assembler is somewhat mechanical and is responsible for for emitting a binary that is to be consumed by the hardware. The resulting binary has two sections, the instructions section which comes first, then the constants section, and finally the variables section. Most of the assembler is fairly straightforward. The assembler can only consume the operators present in Table 3.5. Below is a toy example of how one might go from instructions to binary using the assembler.

```
from maeri.compiler.assembler.states import ConfigForward, ConfigUp
from maeri.compiler.assembler import opcodes
from maeri.compiler.assembler.opcodes import LoadFeatures
from maeri.compiler.assembler.states import InjectEn
from maeri.compiler.assembler.assemble import assemble
from maeri.gateware.compute_unit.top import State
from random import randint, choice

# build out ops
valid_adder_states = [ConfigForward.sum_l_r, ConfigForward.r, ConfigForward.l]
valid_adder_states += [ConfigUp.sum_l_r, ConfigUp.r, \\
    ConfigUp.l, ConfigUp.sum_l_r_f]
valid_mult_states = [InjectEn.on, InjectEn.off]

ops = []

test_state_vec_1 = [choice(valid_adder_states) \\
    for node in range(driver.no_mults - 1)]
test_state_vec_1 += [choice(valid_mult_states) \\
    for node in range(driver.no_mults)]
ops += [opcodes.ConfigureStates(test_state_vec_1)]

test_weight_vec_1 = [randint(-128, 127) \\
    for node in range(driver.no_mults)]
ops += [opcodes.ConfigureWeights(test_weight_vec_1)]
ops += [opcodes.Debug()]
```

```
# returns a list of values to be loaded directly
# into MAERI's memory
binary = assemble(ops, as_bytes=True)
```

CHAPTER 5

RESULTS

The latest scripts as well as older scripts in version control for reproducing the exact results presented in this thesis can be found at the following url:

https://github.com/BracketMaster/reproduce-thesis.

Updates such as precise runtime profiling for the MAERI DNN HW will also be pushed to that repository.

5.1 Compiling and Assembling MNIST

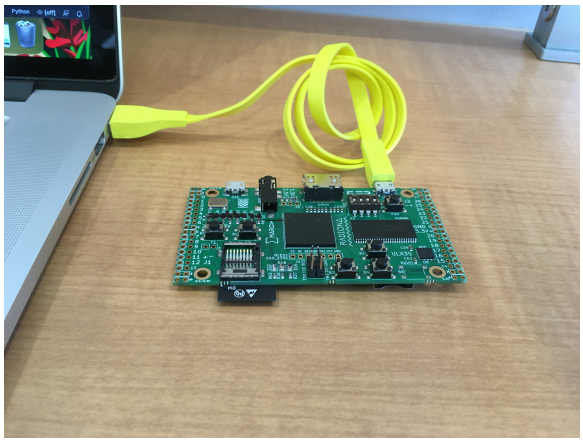

Figure 5.1: FPGA configured with MAERI DNN HW.

The MNIST model presented in Figure 1.4 was compiled into the tensorflow-lite format for the edge-TPU and the ONNX IR format for the Intel Compute Stick 2. The process for running MNIST on MAERI is a little more involved. As mentioned in the previous section, the compiler built for MAERI is currently not fully functional. Since the author is fully able to reason about allocation patterns as well as Reduction-Network configurations, the author wrote a custom Python pre-processor script that uses pre-baked configurations to emit assembly that accomplishes the convolutions

and matmuls specified in the MNIST model. The script must grabs weights from the tensorflow-lite MNIST model originally built for the edge TPU and place these weights in memory primitive that it passes to the assembler to be assembled along with the configuration primitives.

An issue arose when attempting to convert the keras mnist model to the ONNX format which both the Intel compute stick and the thesis accelerator support. Namely, ONNX doesn't support the NHWC (batch size, height, width channel) image format that keras defaults to. To get around this, the MNIST model was trained in both NHWC for the edge-tpu and in NCHW for the Intel Compute stick as well the MAERI accelerator implemented in this thesis. Furthermore, the OpenVino compiler for the Intel Compute Stick does not currently seem to support int8 while the edgetpu compiler does.

5.2 Runtimes

The author was also able to collect precise runtimes for both the Intel Compute Stick and the EdgeTPU as both Google and Intel provide tools to help with this. The author was not able to finish adding support for precise runtime profiling to the MAERI DNN drivers. The author can confidently provide an upper bound of 5ms for the worst case runtime on the MAERI HW. It is possible to determine this by taking the difference between the time when the host issues requests a classification (which is always after the image is sent to the device) and the time when the host recieves a classification. Clearly there is some delay involved in link communications. The author could achieve more accurate runtime profiles by adding a counter register that the host can request to read.

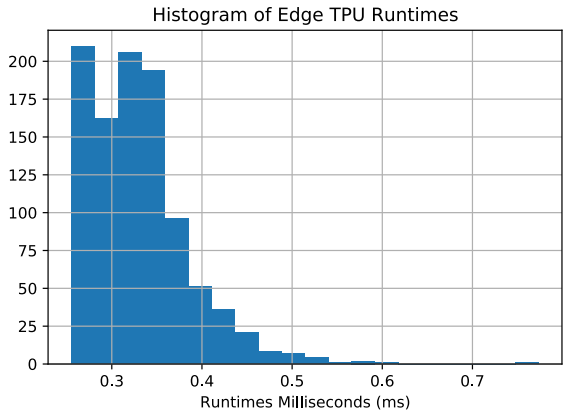

Figure 5.2: Runtime Histogram for the edge-tpu.

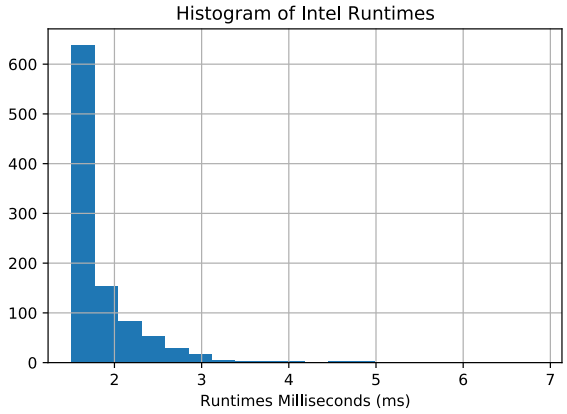

Figure 5.3: Runtime Histogram for the Intel Compute Stick.

5.3 Timing and Resource Consumption

The MAERI DNN hardware comes in at 5919 out of the available 83640 LUTS for a mere 7% of hardware utilization. The MAERI DNN hardware has multiple clock domains which pass at higher frequencies, but are constrained to the frequencies in Table 5.1 to satisfy SDRAM and USB timing requirements.

5.4 Discussion

The Intel Compute Stick only infers on floating point. This may explain the average inference time of .33ms for the edge-tpu being 6 times faster than the 1.88ms

Table 5.1: FMAX for various clock domain.

DOMAIN	MAX	ACTUAL
USB_LOGIC	55.81	48
USB_IO	167.22	12
SDRAM CONTROLLER	120.12	50
COMPUTE CORE	52.55	50

mean inference time for the Intel Compute Stick. The MAERI HW accelerator implemented in this thesis clocks in at 5ms worst case. This is not bad, given the ample room for optimization available in the load-store controllers and the config bus. It is also worth mentioning that in FPGAs are typically notabely slower than their AISC counterparts, in this case, the edge-tpu and the Intel Compute Stick.

CHAPTER 6
CONCLUSION AND FUTURE DIRECTIONS

6.1 Conclusion

Upon discovering an RTL with useful language abstractions, core utilities, and out-of-the-box FPGA support, most of the challenges associated with hardware development disappeared. The most challenging aspect in developing end-to-end custom DNN accelerator hardware is undoubtedly adding support to the compiler stack. Compilers in and of themselves are not necessarily difficult to build, but adding backend support to existing tools can be quite challenging. The author found it impossible to to build MLIR without using Google's custom docker container. In addition, the documentation for the components of MLIR required modification for targeting custom hardware was scant.

GLOW was notably easier to build, but the process for grappling GLOW IR was poorly documented, and the author ran out of time to add any meaningful level of MAERI support to GLOW.

For those desiring to build an end-to-end accelerator as quickly as possible, the author recommends using the ONNX IR as the high-level format to be consumed by the accelerator's compiler. For those with an ample supply of time and patience, the author recommends modifying the GLOW compiler to support their hardware as the GLOW compiler includes useful utilities for performing IR transformations.

6.2 Future Directions

MAERI's performance isn't particularly impressive, but the following optimizations could result in some appreciable speedups:

1. Separate clock domain for compute and memory.

2. OOO support with multiple MAERI reduction networks. Each Maeri redution network becomes a functional unit. A dependency unit schedules the run opcode onto various functional units. The functional units have caches and sit on a mesh interconnect allowing for fused operators.

3. Upgrade to USB 2.0 PHY.

4. Invest in porting to the GLOW compiler framework and take advantage of GLOW's ability to reason about operators and allocations.

6.3 If The Author Could Do It Over

1. The Author struggled excessively writing a custom bus and arbiter for the FPGA. A company by the name of ChipEleven has recently open sourced a formally verified nMIgen AXI arbiter that targets the ULX3s FPGA. The author would just use that.

2. The author tried to start building hardware first. The author should have started by creating the ISA, and then building an ISA level simulator.

REFERENCES

[1] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds., ACM, 2018, pp. 461–475.

[2] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, vol. 12, no. 7, p. 113, 2020.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[4] A. K. I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Proceedings of the IEEE*, vol. 1, pp. 1097–1105, 2012.

[5] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[6] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[7] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[8] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016. arXiv: 1602.07261.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[10] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1800–1807.

[11] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5987–5995.

[12] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[13] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7132–7141.

[14] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8697–8710.

[15] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[16] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[17] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.

[18] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, ACM, 2019, pp. 151–165.

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[20] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S. Liu, and T. Delbruck, "Nullhop: A flexible convolutional neural network accelerator based on sparse

representations of feature maps," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 644–656, 2019.

[21] D. Kim, J. Ahn, and S. Yoo, "Zena: Zero-aware neural network accelerator," *IEEE Design Test*, vol. 35, no. 1, pp. 39–46, 2018.

[22] J. Li, S. Jiang, S. Gong, J. Wu, J. Yan, G. Yan, and X. Li, "Squeezeflow: A sparse cnn accelerator exploiting concise convolution rules," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1663–1677, 2019.

[23] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[24] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 674–687.

[25] *Nvdla primer¶*.

[26] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018. arXiv: 1802.04799.

[27] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018. arXiv: 1805.00907.

[28] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, *Mlir: A compiler infrastructure for the end of moore's law*, 2020. arXiv: 2002.11054 [cs.PL].